EUKLAS: Supporting Integration of Example Code

ABSTRACT

The integration of code snippets into users' target code has not received much attention among researchers. In this paper, we present our new tool, called Euklas, which assists JavaScript programmers with the integration of code snippets. Euklas analyses the original file from where code was copied in order to provide much better "quick fixes" for errors compared with current tools. Euklas also introduces static, heuristic source code checks for JavaScript, even though it is an untyped language. Our evaluation comparing Euklas with the Eclipse Development Environment's JavaScript editor shows that Euklas's user interface is successful and that participants using Euklas were able to fix almost two times more errors and accomplished this in much less time.

Author Keywords

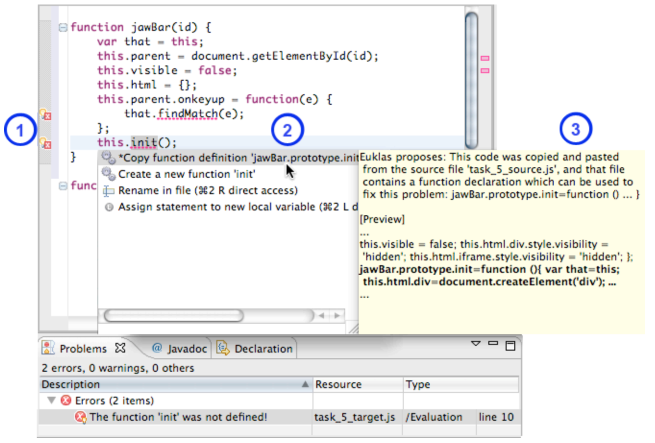
Code Reuse, Copy-and-Paste, JavaScript, Integrating Code, Eclipse, Natural Programming.

ACM Classification Keywords

H.5.2 [Information interfaces and presentation]: User Interfaces—Interaction styles (e.g., commands, menus, forms, direct manipulation). D.2.2 [Software Engineering]: Design Tools and Techniques—User interfaces

INTRODUCTION

Leveraging examples is an established technique in design [13] and has recently received much attention from the research community [1,3,6,7]. With the rise of search engines and web repositories of code along with discussion threads, blogs, and code example websites, people often tailor or "mash–up" parts of existing systems into new systems [17]. Surveys and research show that looking for examples is often people's preferred way to learn to perform a task, or use APIs [4]. Most of the research on reusing examples has focused on building improved search and data mining tools to help with *finding* the examples, targeted at interaction designers [13] and (script) programmers [4,7], which allow them to find code snippets on the web, in APIs [21], and in other code databases [3].



```
function jawBar(id) {
    var that = this;
    this.parent = document.getElementById(id);
    this.visible = false;
    this.html = {};
    this.parent.onkeyup = function(e) {
        that.findMatch(e);
    };
    this.init();
}
func
```

1
2
3
*Copy function definition 'jawBar.prototype.init
Create a new function 'init'
Rename in file (⌘2 R direct access)
Assign statement to new local variable (⌘2 L d
Problems
Javadoc
Declaration

Euklas proposes: This code was copied and pasted from the source file 'task_5_source.js', and that file contains a function declaration which can be used to fix this problem: jawBar.prototype.init=function () ... }

[Preview]

...

this.visible = false; this.html.div.style.visibility = 'hidden'; this.html.iframe.style.visibility = 'hidden'; };
jawBar.prototype.init=function (){ var that=this; this.html.div=document.createElement('div'); ...

...

2 errors, 0 warnings, 0 others

Description	Resource	Type	
Errors (2 items)			
The function 'init' was not defined!	task_5_target.js	/Evaluation	line 10

Figure 1: Euklas enhances Eclipse's JavaScript editor by (1) highlighting errors in the source code, (2) providing quick fixes, and (3) explanations, for copy-and-paste errors based on the code of the used example snippet.

However, there has not been much research on assisting users in *reusing* the examples after they have been found [5,6,15]. Copy-and-paste has been identified as a common usage pattern for reusing code (e.g., blocks or methods) [11,19]. Our aim is to help people with the identification and integration of additional relevant pieces of code to make their copied code work.

This paper introduces our new tool, called EUKLAS (**E**clipse **U**sers' **K**eystrokes **L**essened by **A**ttaching from **S**amples)[1], which helps users to more successfully employ copy-and-paste strategies for reuse. Euklas assists users in fixing errors that are caused by copying and pasting pieces of example code into a target system (see Figure 1).

JavaScript is one of the most popular scripting languages for web programming and is used by a broad variety of users, from end-user developers [14] such as interaction designers [17], to professional programmers. In spite of this widespread use, JavaScript development tools provide less support (e.g., no static code checking, limited auto-completions), compared with other, non-scripting languages [10]. One reason may be that analyzing JavaScript is difficult since it has a weak, dynamic typing that makes it challenging to perform sophisticated static analyses at edit-time.

[1] Euklas is German for Euclase, a gemstone. Euklas is pronounced *oy-class*.

The following example presents a typical use case for the kind of copy-and-paste reuse of JavaScript code that is supported by Euklas. Jamie is a JavaScript developer and has found a snippet showing how to create an enhanced combo-box that she wants to use in her website. However, she is struggling with the integration of the combo-box into her existing codebase. Jamie has to understand the code in more detail to identify which function calls are really necessary and which other pieces of code are relevant to make the code work (e.g. functions, variables, or imports of JavaScript and CSS files). This is usually a time-consuming and cumbersome task especially since Jamie's JavaScript editor does not provide any help. At this point, Euklas can help Jamie with the integration process as it analyzes the target code and inserts *error markers* and squiggly underlines that highlight errors in the code (see #1 in Figure 1). More importantly, Euklas computes quick fixes[2] which would correct these errors (see #2 in Figure 1).

Euklas makes the following three major contributions: **1)** Providing heuristic edit-time source code checks for JavaScript, such as checking for uninitialized variables and undefined functions, which do not require annotations to the code. **2)** Analyzing the example file from where code was copied to provide more detailed error descriptions and much better quick fixes for these errors in the target file. **3)** Using Eclipse's well-established user interface features for marking and fixing errors in a new way, i.e. by helping people to reuse code by employing copy-and-paste strategies.

We evaluated Euklas in a user study with 12 people, comparing it to Eclipse's JavaScript editor. The results showed that the contributions above have been implemented in a usable and effective manner. Participants using Euklas spent much less time integrating example code into their target systems and fixed almost two times as many errors as the control group did.

RELATED WORK

The reuse of example code consists of two main phases: locating the example code and integrating it into the target system. There are several tools that assist users in finding relevant example code (e.g., in repositories, on the web) that they may want to reuse [2,3,9,20]. However, more relevant are tools that support users in integrating code into their systems.

JDA (JAVASCRIPT DATAFLOW ARCHITECTURE) enables users that have no programming skills to mash-up web applications from JavaScript code that was found on the web [15]. It enables users to do this by writing simple HTML commands to connect the different JavaScripts in a way similar to "pipes" in UNIX systems. However, JDA has so far not been evaluated in a user study.

D.MIX targets web designers that are familiar with HTML and scripting languages, such as JavaScript. It enables them to build and share mashups created from pre-existing web sites. d.mix inspired the design of Euklas as it follows a copy-and-paste approach that copies richer representations of the selected data [8]. In that way, elements' parameters can be changed after pasting them to d.mix's editing environment. However, the evaluation showed that participants regarded d.mix as a viable platform for rapid prototyping but not for a larger deployment, due to several issues (e.g., robustness and performance), which Euklas will not suffer from.

The Looking Glass IDE helps middle school students to reuse functionality they find in other programs without requiring them to understand how the code works [6]. It guides students to do this by enabling them to 1) record the execution of the program they are interested in, to 2) identify the start and end of the functionality they are interested in, and to 3) integrate this functionality in their new program.

JIGSAW is a plug-in for the Eclipse IDE that uses a copy-and-paste interaction technique for reusing Java code [5]. Jigsaw inspired the design of Euklas, since it assist developers with the integration of the reused source code into the developer's own source code. It compares the example code and the target code to suggest which pieces of the example code would fit best in the target code. The relevant code is highlighted in Eclipse by using four colors. Conflicting suggestions have to be resolved manually by using a dialog box. However, Jigsaw requires pieces of example and target code that have a similar AST (Abstract Syntax Tree) structure. Jigsaw was tested in a study with two developers, which indicated, "… that conflict resolution is not especially onerous in the cases examined, though usability improvements to our tooling were suggested." [5, p. 224]

EUKLAS

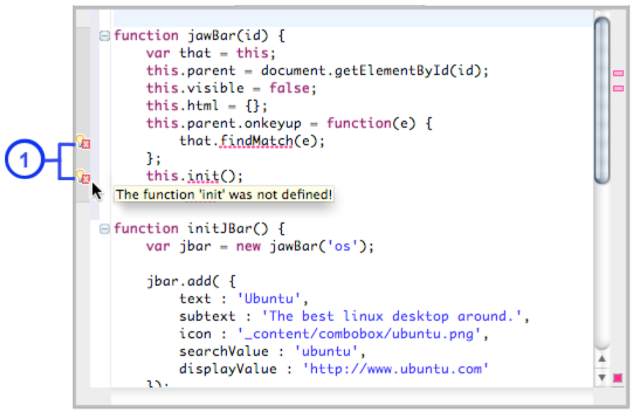
Euklas's copy-and-paste design is strongly inspired by Rosson's and Carrol's study on "the reuse of uses in Smalltalk programming" [19]. They observed reuse strategies where programmers copied and pasted a piece of code (without trying to understand it in detail) that they considered to be promising for the functionality that they intended to reuse. After pasting it to the target context, they let the environment provide editing directions of what would be necessary to make it work. This was often accomplished in several cycles of fixing and waiting for new editing directions, provided by the Smalltalk compiler, until all code was fixed. Euklas's design supports users in employing this reuse strategy. 1) Users copy-and-paste a piece of example code into their target context, and then 2) Euklas marks errors in the code and provides suggestions to fix them (such as to copy-and-paste additional pieces of the example code). These two steps are performed iteratively until all errors have been fixed.

[2] Quick fixes are an Eclipse feature to perform automatic changes in the code if they are selected.

User Interface

Returning to the example discussed in the introduction, we explain how Euklas helps Jamie to integrate the code she found on the web. Jamie discovered that the constructor (`jawBar(id)`, see Figure 2) is probably the essential piece of code for the combo-box she is interested in. She uses Eulkas's "smart copy" and "smart paste" commands to copy-and-paste the constructor into her target file.[3] Both commands can be invoked in four ways in Eclipse's JavaScript and HTML editors. They are available in the right-click context menu, in the main menu, in the toolbar, and as keyboard shortcuts (command + control + C/V, which are similar to the regular copy-and-paste key bindings).

After Jamie pasted the code, Euklas identifies two errors in `jawBar(id)`: the two undefined function calls `findMatch(e)` and `init()` (see #1 in Figure 2). They are undefined because their function definitions are not available in the target file because they have not been copied from the source file. Euklas uses Eclipse's error highlighting feature including the squiggle underlines and the error markers in the margins for indicating these errors in the source code.



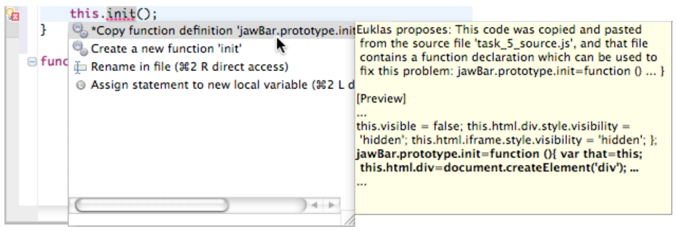
```
function jawBar(id) {
    var that = this;
    this.parent = document.getElementById(id);
    this.visible = false;
    this.html = {};
    this.parent.onkeyup = function(e) {
        that.findMatch(e);
    };
    this.init();

function initJBar() {
    var jbar = new jawBar('os');

    jbar.add( {
        text : 'Ubuntu',
        subtext : 'The best linux desktop around.',
        icon : '_content/combobox/ubuntu.png',
        searchValue : 'ubuntu',
        displayValue : 'http://www.ubuntu.com'
```

Figure 2: Euklas marks errors in the function `jawBar()`, which is the constructor of the enhanced combo-box, after Jamie pasted it into the target file.

In addition to marking the errors in the code, Euklas also suggests solutions for fixing them. By clicking on the error marker that refers to the call of `init()`, Jamie gets four options to fix this error, as shown in Figure 3. Again, Euklas uses Eclipse's functionality for showing the quick fix options. The quick fixes proposed by Euklas (the first two in the list in Figure 3) use the Euklas icon to distinguish them from the quick fixes proposed by Eclipse (the last two in the list). The first quick fix proposed by Euklas, copies the function `jawBar.prototype.init()` which is defined in the example file from which Jaime copied the function `jawBar()`. The second quick fix proposed by Euklas, is a default fix that creates a new function, called `init()` that would have an empty function body. This is sufficient to remove the syntax error, but it would actually not serve Jamie's intention of making the enhanced combo-box work in her website. We added this default option, as it resembles the kind of quick fixes that are offered by Eclipse's Java editor that many people are probably used to.


*Copy function definition 'jawBar.prototype.init
Create a new function 'init'
Rename in file (⌘2 R direct access)
Assign statement to new local variable (⌘2 L d

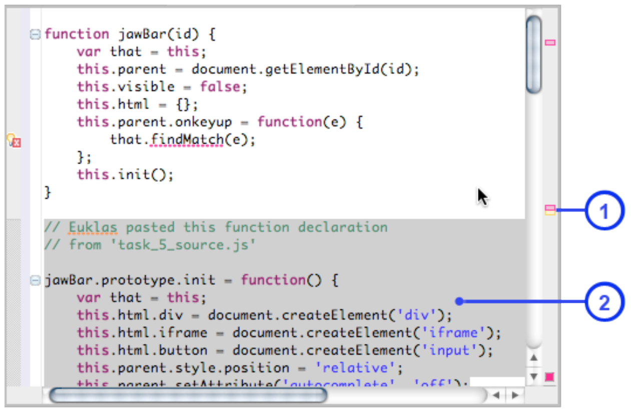
```
    this.init();
}

func
```

Euklas proposes: This code was copied and pasted from the source file 'task_5_source.js', and that file contains a function declaration which can be used to fix this problem: jawBar.prototype.init=function () ... }

[Preview]
...
this.visible = false; this.html.div.style.visibility = 'hidden'; this.html.iframe.style.visibility = 'hidden'; };
jawBar.prototype.init=function (){ var that=this; this.html.div=document.createElement('div'); ...
...

Figure 3: Euklas proposes two quick fixes for the undefined function `init()`. It uses Eclipse's quick fix feature which is well-known by many programmers.



```
function jawBar(id) {
    var that = this;
    this.parent = document.getElementById(id);
    this.visible = false;
    this.html = {};
    this.parent.onkeyup = function(e) {
        that.findMatch(e);
    };
    this.init();
}

// Euklas pasted this function declaration
// from 'task_5_source.js'

jawBar.prototype.init = function() {
    var that = this;
    this.html.div = document.createElement('div');
    this.html.iframe = document.createElement('iframe');
    this.html.button = document.createElement('input');
    this.parent.style.position = 'relative';
```

Figure 4: Euklas pasted the function `jawbar.prototype.init()` from the example file into the target file.

Jamie reads the explanation in the beige window (see Figure 3) that describes how Euklas intends to fix the error based on the example code. She decides that this is the proper fix in her situation and selects it. Euklas pastes the function `jawBar.prototype.init()` from the example file into the target file, as shown in Figure 4 (#2). This successfully fixes the missing definition of the function `init()`. However, the function `findMatch(e)` is still undefined and Euklas identified an additional error in the newly pasted `init()` function (outside the current view, but shown by the marker in the scrollbar: #1 in Figure 4). Jamie continues to fix the remaining errors in the same manner until the code is fixed and the combo-box is working.

Implementation

Euklas was implemented on top of Eclipse's JavaScript editor as a plug-in to the IDE. We selected Eclipse because it provides well-known and well-accepted UI features for finding and fixing errors. However, Euklas's concepts

[3] In the future we will explore how to replace the regular copy/paste commands with ours.

could easily be transferred to other development environments and other languages.

Unlike Eclipse's Java editor, the JavaScript editor does not provide as much programming support. In particular, it only provides error highlighting for syntax errors and some basic quick fixes, such as "rename in file" and "assign statement to new local variable" (see Figure 3). Euklas was designed to help users with programming JavaScript, which also involves partially programming in HTML and CSS.

In general, it is hard to tell which programming constructs definitely lead to runtime errors in JavaScript. Therefore, it has been argued that "… there is no clear-cut definition of what constitutes an 'error'" in JavaScript [10]. Jensen et al. also identify nine potential types of errors of which six are for instance "masked" by dynamic type conversions and undefined values (in some cases programmers may even exploit these behaviors). The other three cause runtime errors: 1) Invoking a non-function value (e.g. undefined) as a function. 2) Reading an undefined variable. 3) Accessing a property which is null or undefined

Kim et al. [11] have studied copy-and-paste strategies in object-oriented programming. They identified that related code snippets (e.g. referenced fields/constants and caller method and callee method) are usually copied together because they belong to the same functionality.

We identified the following list of potential errors that are detected by Euklas in JavaScript and HTML files. Numbers 1-4 are based on the findings of Jensen et al. and Kim et al. and the rest are based on our own experiences with JavaScript.

1. Missing parameter definitions in a function's parameter list
2. Missing local and global variable definitions
3. Missing definitions of functions called normally
4. Missing function definitions of functions being passed as parameters of the global function `setTimeout(Function_Name, Mili_Sec)`
5. Missing CSS style sheet imports (e.g. for general layout definitions)
6. Missing JavaScript file imports
7. Missing HTML elements being accessed by the global function `getElementById (HTML_Element_ID)`

Euklas provides quick fixes for the potential errors 1-4. Unfortunately, Euklas cannot provide quick fixes for the potential errors 5-7 because we have not found a suitable way to make changes in HTML code by using Eclipse's HTML editor. So far, we have not found a way to access the HTML editor's AST (Abstract Syntax Tree) of the HTML parts of the edited file.

In the following we will first discuss the algorithms used for detecting all seven potential errors. Afterwards we will discuss the computation of the quick fixes for the potential errors numbers 1-4.

Euklas employs static analyses on the AST to find potential errors in JavaScript code. The JavaScript AST can either be retrieved from Eclipse's JavaScript editor or from its HTML editor. Euklas's analysis of the AST is invoked if a user "smart pastes" a piece of code from an example file to the target file.[4] As JavaScript was designed to maximize flexibility, it is not strongly typed and has consistency and development style issues [16,18] making it very difficult to analyze the code statically.

Euklas's code analyses are fundamentally different from those that are used in regular static code analysis tools, such as FindBugs [1]. Euklas instead employs heuristic analyses, which has proven to be a successful alternative approach [12]. There are two reasons for this. First, JavaScript does not provide static typing of variables making it impossible to run certain dataflow analyses. Second, we assume that the code of the used examples is syntactically and semantically correct. These two reasons keep our analyses simple, as they relieve us from many checks, such as finding places that could dereference a null pointer, or places in which variables of unrelated types are compared for equality. We consider such analyses as important, but they are not related to errors that are usually caused by copy-and-paste. Euklas uses two different algorithms to identify errors 1-4. The first algorithm checks for undefined variables (# 1,2) and the second algorithm checks for undefined functions (# 3,4).

Undefined variables are detected by using the following error pattern: check for each used variable (e.g., in an assignment, a while loop, and if-statement), and see if it was defined locally, as parameter of the function, in any enclosing function or enclosing function's parameter list, or if it was defined globally. To be able to do this check, Euklas keeps track of the defined variables by using a list of global variables and two stacked lists for the parameters and locally defined variables. This check is significantly eased by the fact that JavaScript uses a simple scoping mechanism. It puts the scopes on a stack and uses the first occurrence of an identifier `x` on the stack as its value.

Undefined functions are detected by using the following error pattern: check for each function call if there is a corresponding function definition. To be able to do this check, Euklas walks the AST twice. In the first run, it checks all function definitions and puts them in a list. In the second run, it checks for each function call to see if it is on the list

[4] In the future, we plan to extend Euklas to search for errors in the entire file. For the current prototype, we focused on the most novel aspect of Euklas, which is using the copy-and-paste information to augment the available corrections.

or not. This check would be fairly easy except that JavaScript has the following constructs. First, functions are objects in JavaScript, making it possible to have constructs like this: `var f = function (x) { return x }`. This makes it more complicated to detect the function name, which would be `f` in this case. Second, JavaScript does not have explicit constructors, but a **`new`** keyword that binds **`this`** to a new object. This allows for instance constructs like in the function `jawBar(id)` that is presented in Figure 2. These make if harder to detect that `this.init()` refers to a function called `jawBar.prototype.init()` and not to a function that is just called `init()`.

Euklas scans all HTML files that are related to the current JavaScript file to identify the potential errors no. 5-7. Euklas knows about the relation between any JavaScript to any HTML file, because it analyzes the imports of JavaScript files in all HTML files that are located in Eclipse's workspace. Based on this, it maintains a bi-directional list of related JavaScript and HTML files. Euklas scans the relevant HTML files for missing JavaScript and CSS imports that have been used in the example HTML file. It also checks to see if a given `HTML_Element_ID` that was used in a call of `getElementById (HTML_Element_ID)`, was defined in any of the related HTML files. The analyses use regular expressions because we were not able to use the HTML editor's AST representation of the HTML code. Errors #5 and #6 are marked with a warning marker ⚠ instead of an error marker, as these errors usually do not lead to any runtime problems. For example, a missing import of a CSS style sheet may cause a strange layout of the website, but does not cause runtime exceptions.

Since the presented analyses are all heuristic, they can generate false positive and false negative results, i.e., they can either mark correct code as an error, or miss marking some existing errors in the code. We consider false negatives to be slightly worse than false positives, as it can be quite cumbersome to find errors compared to being annoyed by error markers that relate to correct code. Therefore we slightly tweaked the heuristics towards minimizing false negatives while increasing the potential of false positives. For example, Euklas reports variables as undefined if they have not been defined explicitly using the **`var`** keyword. This does usually not lead to an error, as in these cases browsers interpret the first assignment of a variable as its definition.

There are some cases in which Euklas produces false positives/negatives because it does not use dataflow analysis techniques. It is, for instance, not able to detect errors that are caused by "hiding" variables. Consider again the example in Figure 2: `var that = this; that.findMatch(e);`. In this case Euklas is not be able to detect that the function call (`that.findMatch(e);`) could be related to the object **`this`** instead of to the object `that`. Therefore, it would produce a false positive, as it does not find the function definition of `jawBar.prototype.findMatch(e)` that belongs to `that`. Another wrong detected case is related to variables that have different types in the source and the target file. This kind of error is particularly difficult to detect with in JavaScript, since reliable type information is only available at runtime.

However, based on our experiences we think that these errors are rather uncommon, although more elaborate analyses may be able to detect them correctly in the future. Jensen et al. [10] have recently published some interesting ideas in this area. They use dataflow analyses and abstract interpretations for inferring type information in JavaScript programs.

Euklas does not only find potential errors in the pasted code, but it does also provide fixes for these errors. Next we explain how fixes are computed based on the example files from which the code was copied. Euklas's "smart copy-and-paste" commands establish links between example files and certain regions in the target file. These links are important because it would be difficult to compute proper fixes for particular errors otherwise (e.g. variable names and functions may be used for different things in the source documents). Each target file can have multiple regions and each region has exactly one link to exactly one corresponding region in an example file from which the code was copied. Euklas maintains the metadata about these connections in memory. It updates the regions in the target file when the file is edited to keep the metadata correct. Euklas loads and saves this data at Eclipse's startup and shutdown. We currently keep this data in a separate file, but we will explore how to save it by using Eclipse's metadata management. Eclipse manages, loads, and saves all error markers automatically. Metadata concerning the available quick fixes is currently only kept in non-persistent memory since it is fast to re-generate whenever it is needed.

Based on the error markers in the target file, Euklas identifies to which region in the target file they belong to find the example file that is connected to this region. Euklas needs to have access to each of the example files, i.e., they have to be present in Eclipse's workspace. We decided that this would be a valid restriction for the first version of the prototype, but we could imagine that the example files could also be external pieces of code (e.g., code on web, external JavaScript files) in the future.

The ASTs of the example files are analyzed for potential solutions to the corresponding errors. For example, if the error refers to an undefined variable, Euklas analyzes the source file for a definition of this variable, under the assumption that the code in the source file is syntactically and semantically correct. If there is a suitable definition, Euklas creates a quick fix proposal that is added to the error marker in the target file. The quick fix proposal consists of a description of the proposed changes (including a preview of how the target code would look after the change), a reference to the example file, and a copy of the relevant AST piece of the example file. In addition to this quick fix, which is based on the example file, Euklas generates additional default solutions. For missing variable declarations, it creates three default solutions: create a new parameter, create a new local variable, and create a new global variable. For missing function declarations Euklas creates one default solution: create a new function definition.

If the user selects a proposed quick fix, Euklas parses the AST of the target file to insert the copied AST piece from the example file. Euklas tries to insert parameters at the same position in the target as in the source. Global and local variables are always inserted as first line in the script or the function declaration, while functions are always inserted last. This is an acceptable strategy for the prototype, but should be changed in the future, based on dependencies in the existing target code.

We had several problems with the implementation of Eclipse's quick fix mechanism in Euklas. First, it turned out the extension points for "quickFixAssistProcessors" and "quickFixProcessors" did not work and we had to use the "markerResolution" extension point instead. Second, we wanted to have nicer looking descriptions in the beige explanation window (see Figure 3) which was not possible. The window's implementation interprets pseudo HTML in which only the tags for "bold" and "line breaks" work. In addition, it is not possible to control the size of the window which makes the code inside sometimes look odd, especially if the lines are long. Third, the insertion of comments in the source of what Euklas has changed (shown in Figure 4) was difficult, as Eclipse's JavaScript AST does currently not support the construction of comment nodes. This leads to two problems. Single lines of code cannot have comments on the same line and sometimes there are arbitrary empty lines between the comment and the code which can lead to nested comments if multiple lines are pasted after each other by Euklas.

EVALUATION

The evaluation of Euklas had the goal of answering the following question: Is the integration of copied and pasted pieces of source code faster and more correct with Euklas than with Eclipse's standard JavaScript editor?

Participants

We feel that Euklas is most appropriate for people who have some experience with using JavaScript, and we did not want to have to train people on how to use Eclipse. Therefore, we recruited participants who had about one year of experience in using Eclipse (for developing in any language), and who had done at least one programming project in JavaScript (using any development environment). We recruited 12 participants (10 male, 2 female) from our local university community. Each of them received a compensation of $15 for participating. Their ages ranged from 19 years to 37 years (median: 25, *s.d* 5). The participants had diverse backgrounds, such as business administration and software engineering. We screened the participants before the study to make sure that their skills met our requirements by requiring them to answer a brief survey with two questions about JavaScript and one question about Eclipse. Almost all participants reported that they program less than one hour in JavaScript per week. Eight of them said that Eclipse is their favorite JavaScript editor, four said Notepad++, and three others either named VIM, Emacs, and GUIM (not mutually exclusive).

Apparatus and Materials

The study was conducted in our lab on the university grounds. We used an iMac (Mac OS 10.5) running a standard Eclipse installation (3.6) including the WTP (Web Tools Project) plug-in. The control group used Eclipse's JavaScript editor, and each experimental group used one of two versions of Euklas: "Euklas lite" and "Euklas full". The main difference between the two versions was that "Euklas lite" only analyzed the target file to identify errors and to compute quick fixes, while "Euklas full" also analyzed the example file(s). We chose to have these two different versions to be able to study the following two aspects separately. First, we wanted to explore the effects of providing error highlighting and quick fixes that are similar to Eclipse's Java editor. Second, we were interested in the effects of having more sophisticated analyses and quick fixes that took the example file(s) into account.

Therefore, Euklas lite identified potential errors and provided the kind of quick fixes that are available in Eclipse's Java editor, such as "create a new variable" and "create a new function". The variables were initialized with a default value representing an empty string[5] and the functions had an empty function body and parameter list. Euklas full had all the features that Euklas lite had, but added additional quick fixes that were based on the analysis of the example file(s). It also used some additional heuristics for detecting errors in the JavaScript and HTML code, i.e., missing CSS style sheet and JavaScript imports that could only be detected by taking the source files into account.

We set up Eclipse's workspace with the relevant target files (a small website consisting of six JavaScript and six HTML files) as well as with the example files (six JavaScript and six HTML files) that were used as tasks of the study. Each of the examples could be executed in Firefox (version 3.6.8) so participants could experiment with its functionality. All examples were taken from the Codeproject[6] website which offers downloadable JavaScript files, which made the study more realistic. However, we reduced the complexity of dealing with the example code by relieving participants from the burden of calling the relevant JavaScript functions inside the HTML files, as Euklas does not provide any help

[5] We cannot provide better default initializations, as variables are not typed in JavaScript.

[6] http://www.codeproject.com/

with accomplishing this. We also copied some parts from the example files to the target files to reduce the number of errors that users had to deal with.

Procedure

The study used a between-subjects design with three conditions: using Eclipse's JavaScript editor and using Euklas in the two different versions. The assignment of the participants to the three groups was done randomly.

Participants in all groups received an oral introduction to the study and signed the consent form. All participants were briefly introduced to Eclipse's JavaScript editor. Members of the experimental groups received an additional introduction to Euklas's extensions to Eclipse's JavaScript editor, i.e. the "smart copy" and "smart paste" menus as well as its error and warning markers, and the quick fixes. We stepped through a short tutorial with all participants to show them what kind of errors they could expect and how to work on the tasks. All task descriptions were located in the code and marked with "TODO" comments. The instructions explained which code should to be copied and what the desired results would be after the code was pasted and all errors were fixed. Participants were instructed that they could use Firefox to view the examples, retrieve additional information from the web, and to test their code after they had integrated the pasted code.

All participants did the tasks in the same order. The tasks were designed to cover all cases in which Euklas provides support as well as cases where it provides misleading help or does not help at all. The difficulty of the tasks increased steadily. This was achieved by raising the number of problems, increasing the number of lines of code of the examples to be pasted, and by making it harder to identify the problems and to fix them. Participants were allowed to work on each task for a fixed amount of time, which varied from 7 to 20 minutes, based on the task's difficulty.

The researcher who conducted the study measured the time it took participants to work on each of the tasks and stopped them if they ran over the maximum time assigned for each task (see Table 1). He also observed participants while they were working and took notes about problems and any comments they made.

Table 1 summarizes the tasks. It shows the number of errors per task, the types of errors included in the tasks (based on the types of errors presented in the "Implementation" section), the lines of code (LOC) of the example files (HTML and JavaScript files together), the total number of lines that had to be copied to solve the tasks, and the maximum time participants were given to complete each of the tasks. At the end of the study, participants filled out a questionnaire. The questions primarily used five-level Likert scales, but some of them were open answer.

Task	# of Errors	Types of Errors*	Source LOC	Copied LOC	Max. Time (min.)
1	1	1	112	10	7
2	3	2, 3	112	13	7
3	2	2, 4	122	29	7
4	3	2, 6, 7	103	5	10
5	5	3	218	125	12
6	10	2, 4, 5	1507	996	20
Sum	**24**		**2174**	**1178**	**63**

Table 1: Summary of the tasks (*types of errors according to the list in the "Implementation" section)

Results

The analysis of the data shows that participants using Euklas full spent on average at least 35% less time[7] (see Figure 5) on integrating the example code into the target system than the control group did. In this calculation, participants who did not complete a task were given the maximum time (shown in Table 1), so the time comparisons are quite conservative—in reality, people would probably take much longer. The data also shows that users in the Euklas full group fixed almost two times as many errors as users in the control group (see Figure 6).

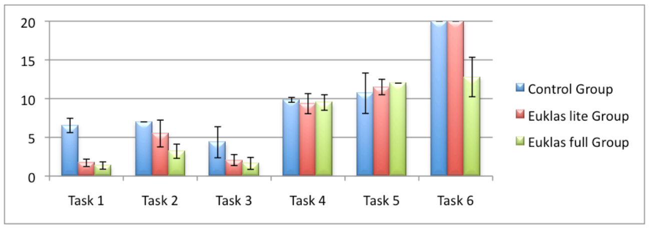


Figure 5: Average time (in minutes) that participants spent on each of the tasks (less is better)

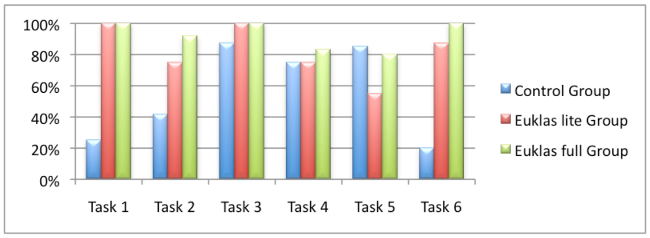


Figure 6: Relative number of fixed errors per task (more is better)

As shown in Table 1, the tasks varied in numbers and types of errors, and maximum time allotted. We analyzed each task separately, rather than calculate an overall metric, which would have required selecting a method to weight each task's contribution to the overall metric. We always used ANOVAs as statistical tests and Tukey post-hoc tests to reveal differences between the different conditions. When reporting means and standard deviation, the results

[7] For tasks it was designed to be helpful for.

will be listed by condition in the following order: control, Euklas lite, and Euklas full.

For task 1, only one participant in the control condition fixed the error, whereas all the participants in both Euklas conditions fixed the error. Due to the differences and lack of normality in the data, no statistical tests were performed. For completion time, there was a significant effect, $F(2,9) = 76.68$, $p < 0.000$. Both versions of Euklas were different from the control condition, but not from each other. The average completion times were 6:32 minutes (*s.d.* 0:56), 1:42 (0:29), and 1:22 (0:28).

For task 2, of the number of errors fixed was significant, $F(2,9) = 4.94$, $p = 0.036$ and Euklas full was different from the control condition. The average numbers of errors fixed were 1.25 (0.96), 2.25 (0.5), and 2.75 (0.5). For completion time, there was a significant effect, $F(2,9) = 11.4$, $p = 0.003$. Euklas full was different from both Euklas lite and the control condition. The average completion times were 7:00 (*s.d.* 0:00), 5:30 (1:44), and 3:12 (0:55).

For task 3, all participants, except one, fixed both errors. That one participant, in the control condition, fixed one of the two errors. Due to the lack of normality in the data, a statistical test was not performed. For completion time, there was a significant effect, $F(2,9) = 5.04$, $p = 0.034$ and Euklas full was different from the control condition. The average completion times were 4:22 (2:00), 2:04 (0:42), and 1:38 (0:46). For task 4 and 5 there were no significant effects for the number of errors fixed and completion time.

For task 6, the number of errors fixed was significant, $F(2,9) = 32.13$, $p < 0.000$. Both versions of Euklas were different from the control condition, but not from each other. The average numbers of fixed errors were 2.0 (2.45), 8.75 (0.96), and 10 (0.0). For completion time, none of the participants in either the control or Euklas lite condition completed the task in less than the 20 minutes allowed. All participants in the Euklas full condition completed the task. The average completion time was 12:48 (2:33). Due to the lack of normality in the data, a statistical test was not performed.

The analysis of the final questionnaires provides more details on the differences between the two Euklas versions. Figure 7 shows participants' perception of the helpfulness of the provided quick fixes. Participants agreed that Euklas full usually provided helpful quick fixes, which is true, as it does not do this in all cases. One Euklas full user nicely expressed why he liked the tool: "Intelligent error messages and debugging makes it infinitely more useful, especially when it checks against the source of your copy." Euklas lite got lower ratings in terms of the helpfulness of its quick fixes. One of the Euklas lite users suggested the following improvement, which reflects exactly the improvements in Euklas full: *"Provide [a] pop-up menu which can suggest to copy blocks of code to resolve errors."*

The questionnaire also asked whether Euklas speeds up the integration of JavaScript code compared to other editors (see Figure 8). Participants using Euklas full strongly agreed with this statement (4.75 out of 5) while participants using Euklas lite did not share this view (3.75 out of 5).

Another item of the questionnaire explored whether participants considered it to be easy to learn the usage of error markers for integrating copied and pasted code (see Figure 9). Again, users of Euklas full agreed to a higher degree (4.75 out of 5) on this question than users of Euklas lite (4 out of 5).

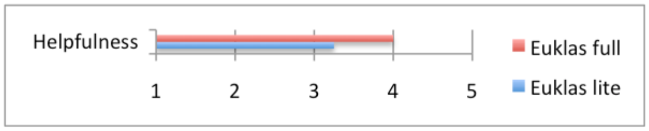


Figure 7: Average helpfulness of Euklas's quick fixes (based on a 5-level Lickert scale—5 is the best)

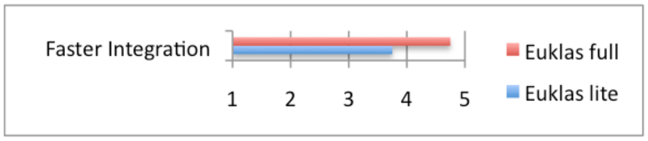


Figure 8: Average perception if Euklas helped to integrate JavaScript code more quickly (based on a 5-level Lickert scale—5 is the best)

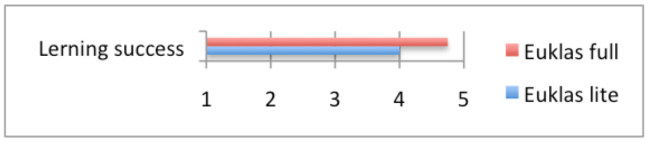


Figure 9: Average perception if the usage of error markers for integrating copied and pasted code was easy to learn (based on a 5-level Lickert scale—5 is the best)

DISCUSSION

We will now discuss each of the tasks in more detail. Task 1 allowed users to familiarize themselves with the system, but even though participants were faster in the Euklas conditions. Almost the same is true for task 2. It contained one error that was a missing function definition. The error was only found by one person in the control condition, and again users of Euklas full were about two times faster. Task 3 had 2 errors, which were found by all persons except one. Even if this was the case, Euklas full users were more than two times faster to completing the task.

Task 4 showed almost no differences in terms of time and number of fixed errors. This was surprising since Euklas full users got help to find a missing external JavaScript import. They got help by a warning maker Euklas put into the HTML file. Nevertheless, many users either ignored the marker or did not see it in the first place, which explains why it took them so long to find this error or why they ran out of time.

Task 5 was designed to prove that Euklas users do not perform worse in cases where the system does not help. The task included a syntax error for which all groups got the same help from Eclipse's JavaScript editor, and a missing function definition that was not detected by either of the two Euklas versions. The time was about the same for all groups, and the number of fixed errors between Euklas full and the control group was also about the same. The much lower number of fixed errors of the Euklas lite group was cause by one participant who did not manage to fix any error.

Task 6 involved the longest and most complex piece of example code, and contained the highest number of errors. In this task, Euklas full played out its advantages. No one in the control group, and only one person in the Euklas lite group completed the task, while all Euklas full users were able to complete it. Users of Euklas full fixed significantly more errors and needed significantly less time to complete the task than users in the other groups.

In sum, the results show that the Euklas lite version, which provided error detection features and standard quick fixes, already brought many improvements in comparison with Eclipse's standard JavaScript editor, which did not offer such features. This is not very surprising, since we know that static analyses and highlighting errors helps. Euklas full, which offered additional analyses and additional quick fixes based on the file the code was copied from, improved Euklas lite even more. It especially showed advantages if the pieces of copied code got longer and/or more complex. This approach of considering a broader context for the computation of potential quick fixes has implications for many other programming languages, such as Java. Similar checks could increase programmers' performance in the same way it did in our case for JavaScript.

Using Eclipse's marker feature for highlighting and fixing errors seems to be an appropriate UI choice as participants had a very positive attitude towards this approach and considered it to be easy to learn. The most important aspect about using this feature is that participants knew what to expect from the provided quick fixes. They knew that these were usually right in Euklas full, but that they could sometimes also be wrong and not helpful. Users were generally able to distinguish between these cases and some manually checked if a proposed quick fix was appropriate or not before they used it.

There were a total of 24 errors that occurred after participants pasted the code from the source files into the target files. Euklas full provided 20 helpful suggestions for fixing these errors, plus 33 generic suggestions (e.g. variable initialized to empty string, function with an empty body) that did not fix the errors. Eclipse added in most cases two additional suggestions per error that were not helpful at all. Participants sometimes picked one of Euklas's unhelpful fixes that created a new function with an empty body (e.g., for one missing function in task 5). They knew that this was not sufficient for fixing the error and always looked at the example code to manually find the right function to fix the error. In cases where Euklas did not show an error at all (e.g., for one missing function in task 5), participants did not perform worse than the participants in the control group.

The evaluation has several limitations. First we did the evaluation with a rather low number of 4 participants in each of the three conditions. Second, participants used the provided tools only for an hour, making it hard for them to provide a reliable feedback. Third, the tasks may not have been representative realistic tasks[8]. The pieces of code were real-world examples that were taken from the web. However, we had to reduce the complexity of dealing with these examples to limit the amount of time spent on completing each of the tasks. Fourth, we did not give participants a very long time to make themselves familiar with the target website or with each of the example snippets as this would have taken too long. This might have biased them to just follow the suggestions without carefully considering what the code was doing.

CONCLUSION AND FUTURE WORK

In this paper we presented our new Eclipse plug-in, Euklas, which helps JavaScript programmers with the integration of example code into their own code. This process often involves copy-and-paste strategies to integrate the functionality in which programmers are interested into their own programs. Euklas supports such strategies by detecting potential errors in the target code and by providing appropriate fixes for these errors, based on the example code. Our evaluation showed that Euklas users were able to fix a much higher number of copy-and-paste errors in much less time than users who used Eclipse's JavaScript editor. One participant summarized his experiences with Euklas in this way: "Copy and pasting code is frowned upon and discouraged within engineering teams. It is better to reuse the code, but copy and pasting does happen and this tool would be useful in those cases."

Euklas has the several advantages in comparison to other JavaScript editors. It is one of the first JavaScript editors to offer error detection mechanisms that are similar to those offered for other languages, such as Java or C++. It uses heuristics to analyze the code and detects, for example, uninitialized variables and undefined functions. Its major contribution is analyzing the file from where code was copied to provide more detailed error descriptions and much better quick fixes for these errors. These analyses help programmers to employ copy-and-paste strategies for integrating example code into their systems. We believe that similar assists could increase programmers' performance in the same way for many other programming languages.

[8] Even if 83% of the participants agreed that the tasks were *realistic 'real-world' tasks*.

A major design advantage of Euklas is that it leverages well-known UI design concepts with which Eclipse users are familiar. It uses Eclipse's marker and quick fix mechanisms for highlighting and fixing errors in the code. Using these mechanisms keeps programmers focused on their code, does not require them to learn new concepts, and reinforces that the proposed quick fixes are not always the solution they are looking for.

In the future, we will implement some additional features. Euklas should provide all the standard checks that Eclipse's Java editor provides, such as fixing misspelled variable names, and fixing the same error in multiple places. Most of these features must employ heuristics, as JavaScript has no type information prior to runtime. We have not implemented these features so far as the work's focus has been on proving that the concept of providing quick fixes for copy-and-paste errors is useful. An interesting idea for providing improved error detections would be to analyze the context of a variable to try to determine its runtime type, which would decrease the number of false positives and negatives. We will also put more effort in making the analysis of related files (e.g. remote JavaScript files and libraries) more elaborate as this will increase the precision of the error analysis and the proposed quick fixes.

Euklas points to a future where programming support tools help developers more by taking into account all of the available contextual information, and the provenance of resources used. The success of Euklas shows that this approach is feasible and can be successful, and developers can make effective use of recommendations, even when they are heuristic.

REFERENCES

1. Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J. and Pugh, W. Using Static Analysis to Find Bugs. *IEEE Software*, *25* (5). 22-29.
2. Bajracharya, S., Ossher, J. and Lopes, C., Sourcerer: An internet-scale software repository. in *ICSE SUITE Workshop '09*, (Vancouver, Canada, 2009), IEEE, 1-4.
3. Brandt, J., Dontcheva, M., Weskamp, M. and Klemmer, S.R., Example-centric programming: integrating web search into the development environment. in *CHI '10*, (Atlanta, GA, 2010), 513-522.
4. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M. and Klemmer, S.R., Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. in *CHI '09*, (Boston, MA, 2009), 1589-1598.
5. Cottrell, R., Walker, R.J. and Denzinger, J., Semi-automating small-scale source code reuse via structural correspondence. in *FSE-16*, (Atlanta, GA, 2008), 214-225.
6. Gross, P.A., Herstand, M.S., Hodges, J.W. and Kelleher, C.L., A code reuse interface for non-programmer middle school students. in *IUI '10*, (Hong Kong, China, 2010), 219-228.
7. Hartmann, B., MacDougall, D., Brandt, J. and Klemmer, S.R., What would other programmers do: suggesting solutions to error messages. in *CHI '10*, (Atlanta, GA, 2010), 1019-1028.
8. Hartmann, B., Wu, L., Collins, K. and Klemmer, S.R., Programming by a sample: rapidly creating web applications with d.mix. in *UIST '07*, (Newport, RI, 2007), 241-250.
9. Holmes, R., Walker, R.J. and Murphy, G.C. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *IEEE TSE*, *32* (12). 952-970.
10. Jensen, S.H., Møller, A. and Thiemann, P., Type Analysis for JavaScript. in *16th International Symposium SAS*, (Los Angeles, CA, 2009), Springer-Verlag, 238-255.
11. Kim, M., Bergman, L., Lau, T. and Notkin, D., An Ethnographic Study of Copy and Paste Programming Practices in OOPL. in *ISESE '04*, (Redondo Beach, CA, 2004), 83-92.
12. Ko, A. and Wobbrock, J., Cleanroom: Edit-Time Error Detection with the Uniqueness Heuristic. in *VL/HCC '10*, (Madrid, Spain, 2010), to appear.
13. Lee, B., Srivastava, S., Kumar, R., Brafman, R. and Klemmer, S.R., Designing with interactive example galleries. in *CHI '10*, (Atlanta, GA), 2257-2266.
14. Lieberman, H., Paternò, F. and Wulf, V. *End User Development*. Springer, Dordrecht, 2006.
15. Lim, S.C.S. and Lucas, P., JDA: a step towards large-scale reuse on the web. in *OOPSLA '06*, (Portland, OR, 2006), 586-601.
16. Mikkonen, T. and Taivalsaari, A., Web Applications – Spaghetti Code for the 21st Century. in *SERP'06*, (Las Vegas, NV, 2008), 319-328.
17. Myers, B., Park, S., Nakano, Y., Mueller, G. and Ko, A., How Designers Design and Program Interactive Behaviors. in *VL/HCC'08*, (Herrsching a. Ammersee, Germany, 2008), 177-184.
18. Richards, G., Lebresne, S., Burg, B. and Vitek, J., An analysis of the dynamic behavior of JavaScript programs. in *PLDI '10*, (Toronto, Canada, 2010), 1-12.
19. Rosson, M.B. and Carroll, J.M. The reuse of uses in Smalltalk programming. *ACM TOCHI*, *3* (3). 219-253.
20. Sahavechaphan, N. and Claypool, K., XSnippet: Mining For sample code. in *OOPSLA '06*, (Portland, OR, 2006), 413-430.

Stylos, J. and Myers, B.A., Mica: A Programming Web-Search Aid. in *VL/HCC '06*, (Brighton, UK, 2006), 195-202.